L5 - Cooperating Objects

January 21, 2020

1 Object Interaction

1.1 Clocks

The next code block is from the *clock-display* example (in Chapter 3).

A digital clock displays time by counting seconds. This is not exactly earth-shattering news. Let's write one in Java, and make use of *abstraction* and *modularity.*

1.1.1 Abstraction

The ability to ignore implementation details in order to focus our attention on a higher-level problem. For example, we need not track every electron through a circuit in order to know what the circuit actually does.

1.1.2 Modularization

Breaking the problem into smaller, well-defined parts, which can be built and tested separately. These parts must interact in well-defined ways. For example, to build a car, we might first build an engine. To build an engine, we might first build a block with cylinders. It is now important that the piston fits within the cylinder.

1.1.3 Back to the Clock

We can use these concepts to build our clock as a *hierarchical modular system.* Do we want to build our clock as

- one monolithic 4-digit clock?
- two two-digit systems?
  - easy to build - only need to figure out the logic once
  - easy to debug and test
  - need some overarching logic to turn into a single clock tho

1.2 The `NumberDisplay` class

This class implements a generic counter with a rollover at a limit specified in the constructor. Calling `increment()` will cause the counter to increase its value by one. Once the count reaches the limit, it will reset to zero.

[15]:

```
/**
 * The NumberDisplay class represents a digital number display that can hold
 * values from zero to a given limit. The limit can be specified when creating
 * the display. The values range from zero (inclusive) to limit-1. If used,
 * for example, for the seconds on a digital clock, the limit would be 60,
 * resulting in display values from 0 to 59. When incremented, the display
 * automatically rolls over to zero when reaching the limit.
 *
 * @author Michael Kölling and David J. Barnes
 * @version 2016.02.29
 */
public class NumberDisplay
{
    private int limit;
    private int value;

    /**
     * Constructor for objects of class NumberDisplay.
     * Set the limit at which the display rolls over.
     */
    public NumberDisplay(int rollOverLimit)
    {
        limit = rollOverLimit;
        value = 0;
    }

    /**
     * Return the current value.
     */
    public int getValue()
    {
        return value;
    }

    /**
     * Return the display value (that is, the current value as a two-digit
     * String. If the value is less than ten, it will be padded with a leading
     * zero).
     */
    public String getDisplayValue()
    {
        if(value < 10) {
            return "0" + value;
        }
        else {
            return "" + value;
        }
    }

    /**
     * Set the value of the display to the new specified value. If the new
     * value is less than zero or over the limit, do nothing.
     */
    public void setValue(int replacementValue)
    {
        if((replacementValue >= 0) && (replacementValue < limit)) {
            value = replacementValue;
        }
    }

    /**
     * Increment the display value by one, rolling over to zero if the
     * limit is reached.
     */
    public void increment()
    {
        value = (value + 1) % limit;
    }
}
```

1.3 The ClockDisplay class

This class contains two instances of the NumberDisplay, one each for the hours counter and the minutes counter. Each of these obviously gets constructed with different rollover limits.

The *composition* of the three objects is accomplished by declaring the two *private objects*, just as we would do with normal fields, in the top of ClockDisplay before the constructor method. These have type NumberDisplay - every class is a unique datatype.

Note that object names are *references*. A side effect of this is that if you set two objects equal to each other, *you will end up with two references to the same object*.

1.3.1 timeTick()

This method wraps around the increment() method of both the NumberDisplay objects within the clock. It is responsible for working out if the the minute counter rolled over and if so, increments the hour counter.

[16]:

```
/**
 * The ClockDisplay class implements a digital clock display for a
 * European-style 24 hour clock. The clock shows hours and minutes. The
 * range of the clock is 00:00 (midnight) to 23:59 (one minute before
 * midnight).
 *
 * The clock display receives "ticks" (via the timeTick method) every minute
 * and reacts by incrementing the display. This is done in the usual clock
 * fashion: the hour increments when the minutes roll over to zero.
 *
 * @author Michael Kölling and David J. Barnes
 * @version 2016.02.29
 */
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private String displayString;    // simulates the actual display

    /**
     * Constructor for ClockDisplay objects. This constructor
     * creates a new clock set at 00:00.
     */
    public ClockDisplay()
    {
        hours = new NumberDisplay(24);  // note that the parameter is passed in here
        minutes = new NumberDisplay(60);
        updateDisplay();
    }

    /**
     * Constructor for ClockDisplay objects. This constructor
     * creates a new clock set at the time specified by the
     * parameters.
     */
    public ClockDisplay(int hour, int minute)
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        setTime(hour, minute);
    }

    /**
     * This method should get called once every minute - it makes
     * the clock display go one minute forward.
     */
    public void timeTick()
    {
        minutes.increment();
        if(minutes.getValue() == 0) {  // it just rolled over!
            hours.increment();
        }
        updateDisplay();
    }
```

```java
    /**
     * Set the time of the display to the specified hour and
     * minute.
     */
    public void setTime(int hour, int minute)
    {
        hours.setValue(hour);
        minutes.setValue(minute);
        updateDisplay();
    }

    /**
     * Return the current time of this display in the format HH:MM.
     */
    public String getTime()
    {
        return displayString;
    }

    /**
     * Update the internal string that represents the display.
     */
    private void updateDisplay()
    {
        displayString = hours.getDisplayValue() + ":" +
                        minutes.getDisplayValue();
    }
}
```

[17]:
```java
ClockDisplay clocky = new ClockDisplay()
```

[18]:
```java
clocky.setTime(11, 58);
```

[19]:
```java
clocky.getTime();
```

[19]:
```
11:58
```

[20]:
```java
clocky.timeTick();
clocky.getTime();
```

[20]:
```
11:59
```

[21]:
```java
clocky.timeTick();
clocky.getTime();
```

[21]:
```
12:00
```

[22]:

```
clocky.timeTick();
clocky.getTime();
```

[22]:

```
12:01
```

1.4 Object names are references

First we create some `int`s:

[24]:

```
int a;
int b;
a = 32;
b = a;
System.out.println(b);
```

```
32
```

Well, that does exactly the right thing that we expected it to do. (Honestly, for Java, that's like the first time ever.)

Now, with a custom class object instead:

[23]:

```
class Person
{
    private String name;

    public Person(String newName)
    {
        name = newName;
    }

    public void changeName( String newName )
    {
        name = newName;
    }

    public String getName()
    {
        return name;
    }
}

Person a;
Person b;
a = new Person("Everett");
b = a; // now b and are the same reference
a.changeName("Delmar");
System.out.println(b.getName());
```

```
Delmar
```

1.5 Object Interaction

Two objects interact when one object calls a method on another object. This interaction is commonly in one direction only - a “client/server” relationship.

In the example, two `NumberDisplay` objects store data on the behalf of `ClockDisplay`:

- `ClockDisplay` is a client object
    - `ClockDisplay` exposes useful services, like actually telling the time
        - these are known as *client methods*
- `NumberDisplay` is a server object
    - `ClockDisplay` only receives data from `NumberDisplay`
    - `ClockDisplay` calls methods that change the fields in `NumberDisplay`
        - these are *server methods*

1.5.1 Constructor and Method Overloading

A class may contain more than one constructor, or more than one method with the same name, so long as each has a distince set of parameter types. A constructor with *no* parameters is called the *default constructor*.

1.5.2 Internal method calls

We can call a method within its own class. These calls do not use dot notation, and are sometimes called *helper methods*. `updateDisplay` in `ClockDisplay` is an example of an internal method; its usage can be seen in `timeTick()`.

If a method is meant to be used in this way, its visibility is often set to `private` so it can only be used inside the class.

[ ]: